

# Transition Semantics

Mattox Beckman  
beckman@iit.edu

Illinois Institute of Technology

- There are many ways we can specify the meaning of an expression. One way is to specify the steps that the computer will take during an evaluation.
- An *evaluation* has the following form:

$$e_1 \rightarrow e_2$$

where  $e$  is some expression, and  $e_2$  is another expression, possibly a value.

Examples:

- if true then 4 else 38  $\rightarrow$  4
- $13 + 4 * 5 \rightarrow 13 + 20$

- Note well:  $\rightarrow$  indicates *exactly one* step of evaluation.

Previous semantics both used proof trees:

- Type derivations* specified the type of an expression.
- Natural semantics* specified the value of an expression.

Today's semantics will use *transitions* to specify the value of an expression. By the end of lecture, you should know how to use transitional semantics to

- determine the value of an expression (i.e., be able to read)
- specify the meaning of a language (i.e., be able to write).

You should also know the Church-Rosser property and be able to give examples of languages that have it and languages that don't have it.

- In transition semantics we need to be able to distinguish between *values* and *expressions*.
  - A *value* is a valid *expression* that can not be evaluated any further.
  - (Note, the converse is not true.)
- Use letters  $U$ ,  $V$ , and  $W$  to represent values.
- Use letters  $M$ ,  $N$ , and  $L$  to represent expressions.

Here are three semantic rules for the `if` statement.

- $\text{if true then } M \text{ else } N \rightarrow M$
- $\text{if false then } M \text{ else } N \rightarrow N$
- $$\frac{L \rightarrow L'}{\text{if } L \text{ then } M \text{ else } N \rightarrow \text{if } L' \text{ then } M \text{ else } N}$$

In English:

- If the conditional part is `true`, evaluate the first branch.
- If the conditional part is `false`, evaluate the second branch.
- Otherwise, if the conditional part is not yet evaluated, evaluate it one step.

- These rules are for the short-circuit version of the boolean operators.

and

$\text{true \&\& } N \rightarrow N$   
 $\text{false \&\& } N \rightarrow \text{false}$

$$\frac{M \rightarrow M'}{M \&\& N \rightarrow M' \&\& N}$$

or

$\text{true || } N \rightarrow \text{true}$   
 $\text{false || } N \rightarrow N$

$$\frac{M \rightarrow M'}{M || N \rightarrow M' || N}$$

- These versions are for the “long” versions.

and

$$\frac{M \rightarrow M'}{M \&\& N \rightarrow M' \&\& N}$$

$$\frac{N \rightarrow N'}{U \&\& N \rightarrow U \&\& N'}$$

$\text{true \&\& } V \rightarrow V$   
 $\text{false \&\& } V \rightarrow \text{false}$

or

$$\frac{M \rightarrow M'}{M || N \rightarrow M' || N}$$

$$\frac{N \rightarrow N'}{U || N \rightarrow U || N'}$$

$\text{false || } V \rightarrow V$   
 $\text{true || } V \rightarrow \text{true}$

- These rules are boring. But we need to include them anyway.

$$\frac{M \rightarrow M'}{M \oplus N \rightarrow M' \oplus N} \quad \frac{N \rightarrow N'}{V \oplus N \rightarrow V \oplus N'}$$

Where  $\oplus$  is  $+$ ,  $-$ ,  $>$ ,  $<$ ,  $\dots$

- These rules are so boring that we don't include them.  
 $0 + 0 \rightarrow 0 \quad 0 + 1 \rightarrow 1 \quad \dots$   
 $1 + 0 \rightarrow 1 \quad 1 + 1 \rightarrow 2 \quad \dots$   
*et cetera...*

**Evaluate:**  $\text{if } 3 > 2 \text{ then } 5 + 9 \text{ else } 2 * 4$   
 $\text{if } 3 > 2 \text{ then } 5 + 9 \text{ else } 2 * 4$   
 $\rightarrow \text{if true then } 5 + 9 \text{ else } 2 * 4$   
 $\rightarrow 5 + 9$   
 $\rightarrow 14$

Another common notation:

$\rightarrow^*$  means “zero or more transitions”, so for example  $3 \rightarrow^* 3$ , and  
 $\text{if } 3 > 2 \text{ then } 5 + 9 \text{ else } 2 * 4 \rightarrow^* 14$

**Evaluate:**  $\text{let } x = 2 + 3 \text{ in let } y = x * x \text{ in } x + y$   
 $\text{let } x = 2 + 3 \text{ in let } y = x * x \text{ in } x + y$   
 $\rightarrow \text{let } x = 5 \text{ in let } y = x * x \text{ in } x + y$   
 $\rightarrow \text{let } y = 5 * 5 \text{ in } 5 + y$   
 $\rightarrow \text{let } y = 25 \text{ in } 5 + y$   
 $\rightarrow 5 + 25$   
 $\rightarrow 30$

**Choice 1 — Call By Value**

$$\frac{N \rightarrow N'}{\text{let } x = N \text{ in } M \rightarrow \text{let } x = N' \text{ in } M}$$

$$\text{let } x = V \text{ in } M \rightarrow [V/x] M$$

$$(\text{fun } x \rightarrow M) V \rightarrow [V/x] M$$

$$\frac{N \rightarrow N'}{(\text{fun } x \rightarrow M) N \rightarrow (\text{fun } x \rightarrow M) N'}$$

**Choice 2 — Call By Name**

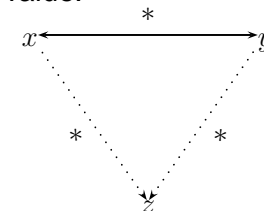
$$\text{let } x = N \text{ in } M \rightarrow [N/x] M$$

$$(\text{fun } x \rightarrow M) N \rightarrow [N/x] M$$

Transition semantics can be thought of as a *term-rewriting system*.  
 Common questions:

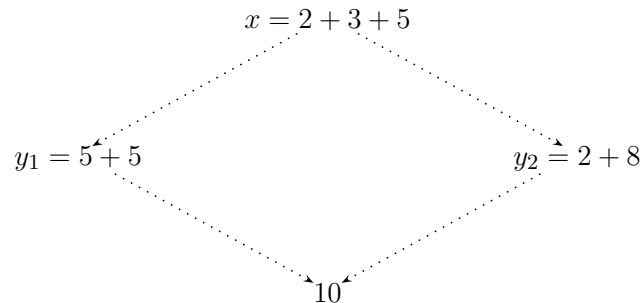
- Does an expression always terminate?
- Can we tell if two expressions are equal?

**Church-Rosser Property:** If  $x \rightarrow^* y$  and  $y \rightarrow^* x$  then  $x$  and  $y$  normalize to the same value.



**Example****§2 Church Rosser Property**

**Confluence:** If  $x \rightarrow y_1$  and  $x \rightarrow y_2$  then  $y_1$  and  $y_2$  normalize to the same value. (Confluence and the Church-Rosser Property coincide.)



This is also known as the “diamond property”

**Problems****§3 Activity**

1. Use transition semantics to calculate the value of  
`let f = fun x -> x * x in f 5`
2. Suppose we add a `match/with` type statement to  $\lambda$ -calculus. Will it still have the Church-Rosser property?
3. Suppose we specify the order of evaluations for arguments in C++. Does that restore the Church-Rosser property?

**Who has it?****§2 Church Rosser Property**

- Alonzo Church and J. Barkley Rosser proved that the  $\lambda$ -calculus has these properties in 1936.
- Very important for theorem provers.
- Most programming languages have this property... some of the time...
- One Benefit: you can check for equality of  $x$  and  $y$  by evaluating them.

Not Church-Rosser

```
1 int i = 10;
2 int sum(int x, int y) { return x + y; }
3
4 cout << sum(i++, i*=2); // 11 + 22 or 21 + 20?
```

**Answers****§3 Activity**

1. Evaluate: `let f = fun x -> x * x in f 5`  
`let f = fun x -> x * x in f 5`  
 $\rightarrow$  `(fun x -> x * x) 5`  
 $\rightarrow$  `5 * 5`  
 $\rightarrow$  `25`
2. Suppose we add a `match/with` type statement to  $\lambda$ -calculus. Will it still have the Church-Rosser property? **Yes**
3. Suppose we specify the order of evaluations for arguments in C++. Does that restore the Church-Rosser property? **No**

```
1 char *i = new char('x');
2 cout << (int) i; // different each time
```